

1. 本周工作

阅读《C++ concurrency in action》Chapter 3. Sharing data between threads（内容提炼见附录）；

参加CNCC.

2. 下周工作

2.1

阅读Chapter 4. Synchronizing concurrent operations, 进行相关编程实践；

2.2

我这与姜老师有关的指导性文件有：

- 面向超分辨率图形系统的应用迁移技术与示范-v3
- 20150429浙大大屏项目沟通会议纪要

老师那可有一些新的材料，让我去了解一下其他主题。

附录：

Chapter 3. Sharing data between threads

3.1 Problems with sharing data between threads

引入**竞态**：当两个或多个进程访问（读、写）共享的数据时，若程序结果受这它们的访问顺序影响，此时将出现竞态。

在10月11日周报，报告《Modern Operating Systems》(4th Edition) -- Chapter 2. Process and Thread时，也提到了竞态。

3.2 Protecting shared data with mutexes

mutex用来实现对多个线程对临界区的互斥（**mutual exclusion**）。

3.2.1 Using mutexes in C++

mutex类有lock和unlock成员方法，但不推荐直接使用这种低层的API.

C++库提供了形如第二章中thread_guard的lock_guard, 实现了RAII原则，在被创建时进行加锁，在析构时进行解锁 —— 避免了线程意外退出，而未能解锁的情况。

3.3.2 Structing code for protecting shared data

举出了一个通过指针或引用，将临界区内保护的数据传递到临界区外，未能保护共享数据的例子。

“

Don't pass pointers and references to protected data outside the scope of the lock, whether by returning them from a function, storing them in externally visible memory, or passing them as arguments to user-supplied functions.

这种行为，在《Java并发编程实战》描述“发布与逸出”的章节3.2也有谈及。

3.2.3 Spotting race conditions inherent in interfaces (发现接口中的内在竞态)

考虑支持下列五种操作的stack:

- bool empty()
- size_t size()
- T& top();
- T const& top() const; // 重载
- void push(T const&);
- void pop();

提出了在stack类中复合操作的竞态(race condition):

- 先检查再操作，empty()与top()间的竞态:

```
if (!s.empty())
{
    int const value = s.top();
}
```

假设栈s只有一个元素。对于线程A, 在检查empty()后，若另一个线程B执行了pop()操作，那么线程A对一个空栈执行pop()操作是一个未定义的行为(undefined behavior)。

- top()与pop()间的竞态:

```
if (!s.empty())
{
    int const value = s.top();
    s.pop();
    do_something(value);
}
```

假设栈s有两个元素。考虑线程A, B同时执行完了s.top(), 都获得栈顶的元素, 之后它们都执行s.pop(). 这样将导致栈顶元素被访问了两次, 而另一个元素还未来得及访问就被弹出了。

针对这种竞态, 有人提出将top()和pop()操作合为一个原子操作, 由于**可能抛出异常的构造函数**的存在, 这将引入一个新问题。实现stack<vector>对象的拷贝:

```
stack<vector<int>> s2(s1);
```

此时若s1.pop()出了一个元素, s2由于申请内存失败, 在构造器中抛出了bad_alloc异常, 导致了元素在弹出后丢失的情况。

选择1: 通过引用传递 (Pass in a reference)

```
vector<int> result;
some_stack.pop(result);
```

选择2: 拷贝构造和移动构造器中不允许抛出异常 (Require a no-throw copy constructor or move constructor)

选择3: 返回一个指向弹出元素的指针 (Return a pointer to the popped item)

- 优点: 不会抛出异常
- 缺点: 需要管理分配给指针所指向元素的内存。对于简单类型, 管理内存花去的成本会超过按值返回的方式。

选择4: 同时实现选择1, 以及选择2、3中的一种。

3.2.4 Deadlock: the problem and a solution

假设一个任务的执行需要同时获得锁L1和锁L2. 线程P1, P2都执行这个任务:

- P1获得了锁L1, 等待获得L2
- P2获得了锁L2, 等待获得L1

这两个线程互相等待, 形成了死锁。

经典的解决方案之一, 是在不同线程中都按特定顺序加锁。

同时，C++标准库提供了可以同时lock多个mutex的方法：

```
lock(lhs.m, rhs.m);  
lock_guard lock_a(lhs.m, std::adopt_lock);  
lock_guard lock_b(rhs.m, std::adopt_lock);
```

3.2.5 Further guidelines for avoiding deadlock

提出了多种在多线程编程中，防止死锁的指导意见。

3.2.6 Flexible locking with std::unique_lock

提出了一种不同于3.2.4中，用于**延迟**“同时lock多个mutex”操作的类 —— unique_lock：

```
unique_lock<mutex> lock_a(lhs.m, std::defer_lock);  
unique_lock<mutex> lock_b(rhs.m, std::defer_lock);  
lock(lock_a, lock_b);
```

由于lock_a构造函数的第二个参数为defer_lock, 它在构造时不会lock lhs.m, 在第三行才会获得这个lhs.m锁。(lock_b同)

3.2.7 Transferring mutex ownership between scopes (在不同代码域中转换thread的所有权)

3.2.8 Locking at an appropriate granularity (以适合的粒度加锁)

3.3 Alternative facilities for protecting shared data

用于保护共享数据的可选工具。

mutex不是唯一用于保护共享数据的工具，其它一些工具被引入用于在一些特定场景中保护共享数据。

考虑对只需要在创建时进行保护的共享数据。这种场景可能发生在：

- 在创建后只读的数据 (read-only once created)
- 数据提供的API都隐式地实现了对自身的保护（同步读写保护）

3.3.1 Protecting shared data during initialization

惰性初始化

在单线程编程中，对于初始化代价较高的数据类型，可以选择延后它的初始化，在被使用时才初始化它 —— **惰性初始化(lazy initialization)**

```
shared_ptr<some_resource> resource_ptr;
if (resource_ptr == nullptr)
{
    resource_ptr.reset(new some_resource);
}
resource_ptr->do_something();
```

多线程中的惰性初始化

在多线程编程中，由于多个线程可能同时调用`reset()`方法，引起一次或多次不必要的`reset()`，所以我们需要实现临界区互斥：

```
shared_ptr<some_resource> resource_ptr;
mutex resource_mutex;

void foo()
{
    unique_lock<mutex> lk(resource_mutex);
    if (resource_ptr == nullptr)
    {
        resource_ptr.reset(new some_resource);
    }
    lk.unlock();
    resource_ptr->do_something();
}
```

这样带来的新问题是：即使在完成了`resource_ptr`的初始化后，每个调用访问`foo`函数进行`resource_ptr->do_something()`的线程，都得等待地去检查`resource_ptr`，这样降低的并发性。

尝试解决并发性下降问题的双次检查加锁模式

许多人尝试解决这个问题，其中有一个臭名昭著的(infamous)解决方法是**双次检查加锁 (Double-Checked Locking)**模式：

```
void undefined_behaviour_with_double_checked_locking()
{
    if (resource_ptr == nullptr)           // if1
    {
        lock_guard<mutex> lk(resource_mutex);
        if (resource_ptr == nullptr)       // if2
        {
            resource_ptr.reset(new some_resource);
        }
    }
    resource_ptr->do_something();
}
```

if1 的目的是解决上述由于"每个调用访问foo函数进行resource_ptr->do_something()的线程, 都得等待地去检查resource_ptr"引发的并发性下降的问题:

- 理想的, 在resource_ptr非空时, if内的语句块不被执行, 不会引发锁的等待或持有。

if2 的目的是保证resource_ptr只被reset一遍。假设多个线程 (P1, P2) 同时进入if1, P1先获得锁, 那么程序逻辑应该如此:

1. P1继续执行, 完成了reset()
2. 由于互斥锁的存在, P2要在P1完成reset并退出if1后, 才可能获得resource_mutex继续执行。当它执行if2时, 看到的resource_ptr已被reset, 故而实现了resource_ptr只被重置一遍的要求。

不幸的是, 这种模式有一个致命的错误 —— 在临界区外访问了共享数据。

- 在if2(临界区)内, 线程可以reset resource_ptr
- 在if1判断语句(临界区)外, 线程对resource_ptr进行了判断

这带来的问题是, 临界区外的读操作(线程P1)与另一个线程在临界区内的写操作(线程P2)并不同步:

- 即使P1看到了resource_ptr值发生了改变, 也未必能看到它所指向的新创建的实例。

C++标准将这种竞态称之为**数据竞态**, 也是未定义行为中的一种。第五种详细讨论了内存模型, 其中包含了形成数据竞态的原因。

once_flag类型和call_once方法的引入

为实现惰性初始化, 同时保证并发性, C++11引入了once_flag类型及call_once方法:

```
shared_ptr<some_resource> resource_ptr;
once_flag resource_flag;

void init_flag()
{
    resource_ptr.reset(new some_resource);
}

void foo()
{
    call_once(resource_flag, init_resource);
    resource_ptr->do_something();
}
```

static局部变量的多线程同步问题

在C++11之前的编译器中, 当多个线程进入函数体时, 它们都可能认为自己是第一个调用这个函数线程, 进而导致static变量的多次赋值。

这种问题在C++11编译器中得到了解决。

总结：数据唯独在初始化中需要保护的这种应用场景，只是“极少（难得）被更新的数据”应用场景中的一种特例。

3.3.2 Protecting rarely updated data structures

考虑DNS服务器：

- 在大部分时间里，DNS entry不被更新，因此entry可以被各个线程共享（读共享）
- 在小部分时间里，DNS entry会被更新，此时要实现读写互斥。

为实现这种读共享、读写互斥，有一项在boost中被使用，但并未被C++11通过的读写锁（reader-writer mutex）

3.3.3 Recursive locking

如果一个线程尝试去加锁一个它已经持有的mutex，这将导致未定义的行为。

recursive_mutex支持多次加锁的操作：

- 在recursive_mutex被另一个线程P2加锁前，原拥有recursive_mutex的线程P1必须解锁
- 在一个线程中，加多少次锁，就需要解多少次锁